Microsoft Research

2012 Summer School on Concurrency

August 22–29, 2012 | St. Petersburg, Russia

Introduction to Parallel Programming

Section 4.

Parallel programming language Chapel

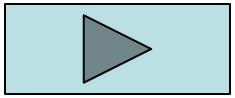
Victor Gergel, Professor, D.Sc.
Lobachevsky State University of
Nizhni Novgorod (UNN)

Contents

- Parallel programming language Chapel
  - Distributed memory systems – MPI technology
  - Shared memory systems – OpenMP technology
  - The model of partitioned global address space (PGAS)
  - Chapel language
    - Language elements
    - Task parallelism
    - Extended capabilities for data processing
    - The concept of executor (*locale*)
    - Data distribution management
    - Examples

Approaches to parallel programs development

- **Usage of libraries** for existing programming languages – MPI for C and Fortran for distributed memory systems
- **Usage of "above-language" means (directives, comments)** - OpenMP for C and Fortran for shared memory systems
- **Extensions of existing programming languages** – e.g. **CAF**, UPC
- **Creation of new – parallel – programming languages** – e.g. **Chapel**, X10, Fortress,…

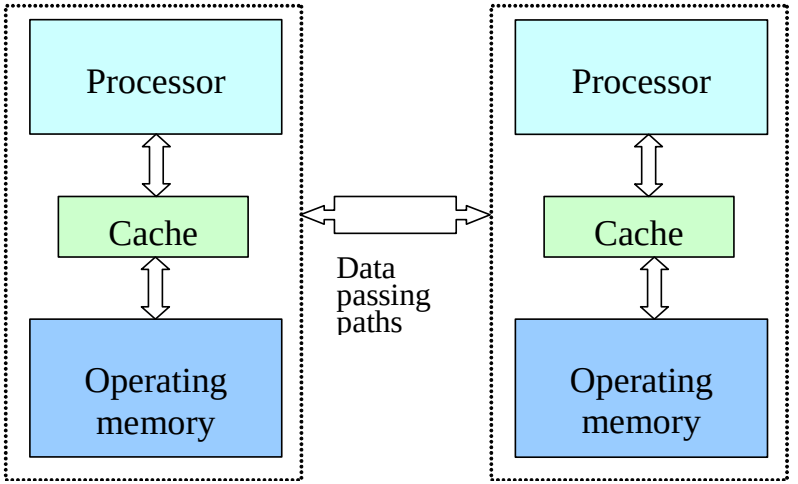


Development of parallel programs for distributed memory systems: MPI…

In computing systems with distributed memory processors work independently from each other.

For parallel computing organization one needs to:

- *Distribute* computational load,
- *Organize* informational interaction (*data passing*) between processors



Solution for all the above items is provided by MPI (message passing interface)

Development of parallel programs for distributed memory systems: MPI…

- Within MPI a single program is developed for a problem solution and is run simultaneously on all available processors
- The following capabilities are available for arrangement of different computations on different processors:
  - Possibility to use different data for a program on different processor,
  - Ways to identify the processor on which the program is being executed
- Such way of parallel computations organization is usually called “*single program multiple processes” (SPMP*)

Development of parallel programs for distributed memory systems: MPI…

- MPI offers plenty of data transfer operations:
  - Provides different ways of sending data,
  - Implements practically all main communication operations.

These capabilities are the strongest advantage of MPI (the name of the technology – Message Passing Interface – speaks for itself)

Development of parallel programs for distributed memory systems: MPI…

What is MPI?

- MPI is the standard which must be met by message transfer set-ups.
- MPI is the software which provides message transfer capability and meets all the requirements of the MPI standard:
  - The software must be arranged as programming modules libraries (*MPI library*),
  - The software must be available for the most widely used algorithmic languages C and Fortran.

Development of parallel programs for distributed memory systems: MPI…

MPI advantages

- MPI allows to significantly mitigate the problem of parallel programs portability
- MPI helps increase parallel computations efficiency – there are MPI library implementations for nearly all types of computing system
- MPI decreases complexity of parallel programs development:
  - Bigger part of main data transfer operations are provided by MPI standard,
  - There are many libraries of parallel methods which use MPI.

MPI: main concepts and definitions…

Parallel program notion

- A *parallel program* in MPI is a set of simultaneously executed *processes:*
  - Each process of a parallel program is created based on a copy of same source code (*SPMP model*),
  - Processes can run on different processors ; at the same time one processor can host several processes.
- Original source code is developed using algorithmic languages C or Fortran and MPI library.
- The number of processes and the number of processors to be used are defined by MPI programs execution environment at the program start. All processes of a program are enumerated. Process number is called process ***rank***.

MPI: main concepts and definitions…

Four main concepts underlie the MPI technology:

- Message transfer operations
- Types of data in messages
- Communicator concept (*group of processes)*
- Virtual topology concept

MPI: main concepts and definitions…

Data transfer operations

- Message transfer operations are the MPI fundamentals.
- MPI provides different types of functions:
  - Pared *(point-to-point)* operations between two processes,
  - *Collective* communication actions for simultaneous cooperation of several processes.

MPI: main concepts and definitions…

Communicators concept…

- A *communicator* in MPI is a specially created service object which joins a group of processes and a set of additional parameters (*context*):
  - Paired operations of data transfer are executed for processes belonging to same communicator,
  - Collective operations are applied simultaneously for all processes of a communicator.
- Specifying the communicator in use is mandatory for data transfer operations in MPI.

MPI: main concepts and definitions…

Communicators concept

- New communicators can be created and existing can be destroyed during computations.
- Same process can belong to different communicators.
- All processes of a parallel program are included in a default communicator with MPI_COMM_WORLD identifier.
- To arrange data transfer between processes from different groups one needs to create a global communicator (*inter-communicator*).

MPI: main concepts and definitions…

Data types

- It is required to indicate type of transferred data when specifying sent or received data in MPI functions.
- MPI contains a big set of *base data types* very similar to data types in algorithmic languages C and Fortran.
- MPI offers ways to create new *derived* data types for more exact and laconic specification of transferred messages contents.

MPI: main concepts and definitions…

Virtual topology

- Logical topology of connection lines between processes has the structure of a complete graph (regardless of existence of actual physical communication links between processors).
- MPI offers a capability of describing a set of processes as a grid of arbitrary dimension. Boundary processes of grids can be claimed adjacent therefore torus-type structures can be defined based on grids.
- MPI also has means of forming logical (virtual) topologies of any desired type.

MPI: Sample program

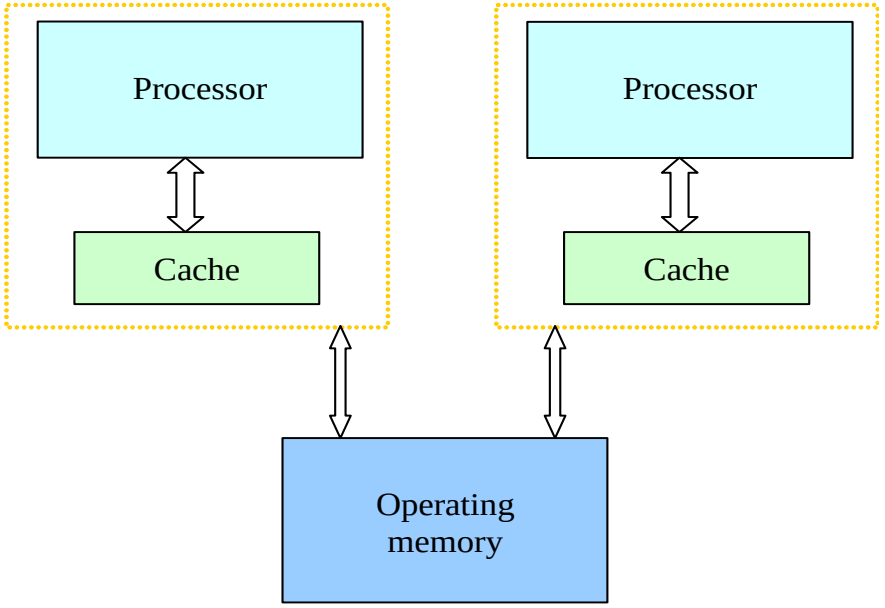
```
#include " mpi.h "
int main(int argc, char* argv[]) {
  int ProcNum, ProcRank, RecvRank;
  MPI_Status Status;
    MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  if ( ProcRank != 0 )
      // Actions for all processes except for process 0
      MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
    else { // Actions for process 0
      printf ("\n Hello from process %3d", ProcRank);
      for ( int i=1; i < ProcNum; i++ ) {
        MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
        printf("\n Hello from process %3d", RecvRank);
      }
  }
  // Job finishing
  MPI_Finalize();
  return 0;
}
```

MPI: additional information

1. Gergel, V.P. Theory and practice of parallel computing. – M.: Binom. The Knowledge Laboratory, Internet University of Information Technologies, 2007.– 424 pp. (In Russian)
2. Antonov, A.S. Parallel programming using MPI technology: Tutorial. –M.: MSU publishing house, 2004.- 71 pp. (In Russian)
3. Nemnyugin, S.A., Stesik, O.L. Parallel programming for multi-processor computer systems.- BHV – Petersburg, 2002, 400 pp.

Parallel programs development for shared memory systems: OpenMP…

OpenMP interface is intended to be a parallel programming standard for multi-processor systems with shared memory (SMP, ccNUMA,…)

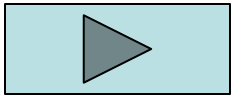


In general case shared memory systems are described as a model of a parallel computer with random access to memory (*parallel random-access machine – PRAM*)

Parallel programs development for shared memory systems: OpenMP…

Fundamentals of the approach

- To indicate the possibility of parallelization developer inserts directives or comments in the source code.
- Program's source code remains unchanged – the compiler can ignore added indications and thus build a regular serial program
- The compiler which supports OpenMP replaces the parallelism directives with some additional code (usually in a form of some parallel library functions calls)



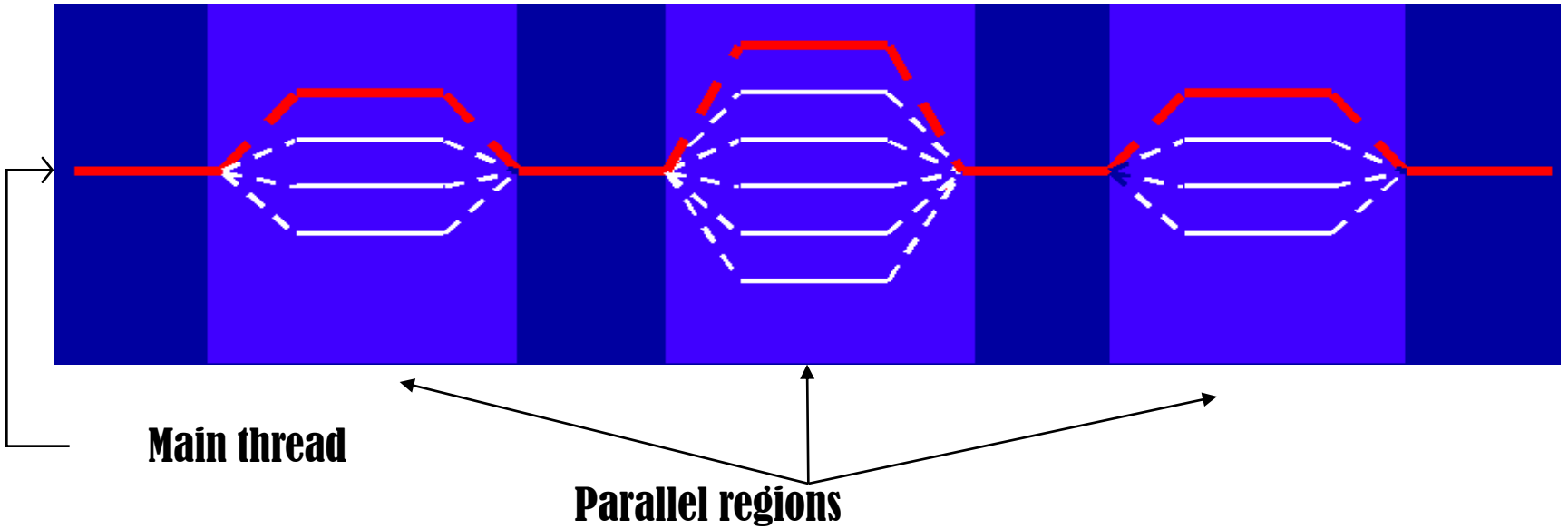
Parallel programs development for shared memory systems: OpenMP…

Reasons for reaching the effect – shared by parallel processes data reside in shared memory, no message passing operations are needed to arrange the communication.

Parallel programs development for shared memory systems: OpenMP…

Parallelism arrangement principle…

- Usage of threads (common address space)
- Pulse (fork-join) parallelism



Parallel programs development for shared memory systems: OpenMP…

Parallelism arrangement principle…

- When executing the normal code (outside of parallel regions) the program is executed by one thread (*master thread*)
- When directive #**parallel** appears the thread team is created for parallel execution of computations
- When leaving the region of #parallel directive effect synchronization is done, all threads except for the master thread are destroyed
- Serial execution of code continues (until the next occurrence of directive #parallel)

Parallel programs development for shared memory systems: OpenMP…

Advantages:

- Step-by-step (incremental) parallelization
  - Possible to parallelize serial programs step-by-step, not changing their structure
- Single source code
  - No need to support serial and parallel variants of a program as directives are ignored by regular compilers (in general case)
- Efficiency
  - Taking into account and making use of shared memory systems resources
- Portability, support in most widespread languages (C, Fortran) and platforms (Windows, Unix)

Parallel programs development for shared memory systems: OpenMP…

Example – Sum of vector elements

```
#include <omp.h>
#define NMAX 1000
main () {
  int i, sum;
  float a[NMAX];
  <data initialization>
  #pragma omp parallel for shared(a) private(i,j,sum)
  {
    sum = 0;
    for (i=0; i<NMAX; i++)
      sum += a[i];
    printf ("Sum of vector elements is equal %f\n«,sum);
  } /* End of parallel region */
}
```

Parallel programs development for shared memory systems: OpenMP…

Additional information:

1. Gergel, V.P. High performance computing for multiprocessor multicore systems. – M.: MSU publishing house, 2010. (In Russian)
2. Antonov, A.S. Parallel programming using OpenMP technology: Tutorial. – M.: MSU publishing house, 2009.- 77 pp. (In Russian)
3. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Melon, R.. Parallel Programming in OpenMP. San-Francisco, CA: Morgan Kaufmann Publishers., 2000.

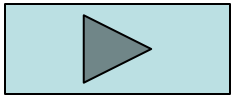
Parallel programs development based on PGAS model…

HPCS Program: High Productivity Computing Systems (DARPA)

Goal: 10x productivity increase by 2010

(Efficiency = Performance + Programmability + Portability + Reliability)

- **Phase II**: Cray, IBM, Sun (July 2003 – June 2006)
  - Existing architectures analysis
  - Three new programming languages (Chapel, X10, Fortress)
- **Phase III**: Cray, IBM (July 2006 – 2010)
  - Implementation
  - Work continued for all proposed languages



Parallel programs development based on PGAS model…

General overview of the HPCS program languages:…

- New object-oriented languages which support a wide set of programming means, parallelism and safety.
- Global name space, multithreading and explicit mechanisms to work with locality are supported.
- Existing means for arrays distribution among computational nodes. Existing means for linking execution threads with data processed by those threads.
- Both data parallelism and task parallelism are supported.

Parallel programs development based on PGAS model…

General overview of the HPCS program languages:…

Locality management

- All three languages provide user access to virtual units of locality called *locales* in Chapel, *regions* in Fortress and *places* in X10.
- Each execution of a program is bundled to a set of such locality units which are mapped by operating system to physical entities like computational nodes.
- This provides users with a mechanism for

  (1) data collections distribution among locality units,

  (2) balancing different data collections and

  (3) linking threads with data they are processing.

Parallel programs development based on PGAS model…

General overview of the HPCS program languages:…

- **Fortress** and **X10** provide extensive libraries of built-in distributions. These languages also provide a possibility to create user-defined data distributions via index space decomposition or combining distributions in different dimensions.
- **Chapel** doesn't provide built-in distributions but provides an extensive infrastructure which supports arbitrary user-defined data distributions capable of sparse data management.

Parallel programs development based on PGAS model…

General overview of the HPCS program languages:…

Multithreading (loop parallelization).

- **Chapel** distinguishes serial loops “for” and parallel loops “forall” in which iterations on index region elements are performed without limitations. Users are responsible for avoidance of dependencies which can lead to data races.
- In **Fortress** the “for” loop is parallel by default thus if loop iterations are performed on a distributed dimension of an array these iterations will be grouped by processors according to data distributions.
- **X10** has two types of parallel loops: “foreach” which is limited to a single locality unit and “ateach” which allows to perform iterations on several locality units.

Parallel programs development based on PGAS model

General overview of the HPCS program languages:

- **Chapel** - http://chapel.cray.com/
- **Fortress -** http://fortress.sunsource.net/
- **X10** - http://x10-lang.org/

Parallel programs development based on PGAS model: Chapel…

Preview – «Hello, world» program

- Fast prototyping

```
writeln("Hello, world");
```

- Structured programming

```
def main() {
  writeln("Hello, world");
}
```

- Application

```
module HelloWorld {
  def main() {
    writeln("Hello, world");
  }
}
```

Parallel programs development based on PGAS model: Chapel…

Language Overview:

- **Syntax**
  - Uses some features of C, C#, C++, Java, Ada, Perl, ...
- **Semantics**
  - Imperative, block-structured, arrays
  - Object-oriented programming (optional)
  - Static typing

Parallel programs development based on PGAS model: Chapel…

Language elements - *ranges*:

- **Syntax**

  range-expr:

  [low] .. [high] [by stride]

- **Semantics**

  Regular sequence of integer values

  – *stride > 0: low, low+stride, low+2*stride, ... ≤ high*

  – *stride < 0: high, high+stride, high+2*stride, ... ≥ low*

- **Examples**

  – 1..6 by 2 // 1, 3, 5

  – 1..6 by -1 // 6, 5, 4, 3, 2, 1

  – 3.. by 3 // 3, 6, 9, 12, ...

Parallel programs development based on PGAS model: Chapel…

Language elements - *Arrays*:

- **Syntax**

  array-type:

  [*index-set-expr*] *type*

- **Semantics**

  Array of elements, size is defined by a set of indexes

- **Examples**

```
– var A: [1..3] int,          // Array of 3 elements
      B: [1..3, 1..5] real, // 2D array
      C: [1..3][1..5] real; // Array of arrays
```

Parallel programs development based on PGAS model: Chapel…

Language elements - *Loop*:

- **Syntax**

  for-loop:

  for index-expr in iterator-expr { stmt-list }

- **Semantics**

  Body expression evaluation on each iteration of the loop

- **Example**

```
– var A: [1..3] string = (“DO“, “RE“, “MI“);
– for i in 1..3 do write(A(i));          // DOREMI
– for a in A { a += “LA“; write(a); } // DOLARELAMILA
```

Parallel programs development based on PGAS model: Chapel…

Language elements – *Conditional Expressions*:

- **Conditional Expression**

```
if cond then computeA() else computeB();
```

- ***while* loop**

```
while cond {
  compute();
}
```

- **Select expression**

```
select key {
  when value1 do compute1();
  when value2 do compute2();
  otherwise compute3();
}
```

Parallel programs development based on PGAS model: Chapel…

Language elements - *Functions*:

- **Example**

```
def area(radius: real)
  return 3.14 * radius**2;

writeln(area(2.0)); // 12.56
```

- **Example of a function with default values of arguments**

```
def writeCoord(x: real = 0.0, y: real = 0.0) {
  writeln("(", x, ", ", y, ")");
}
writeCoord(2.0);   // (2.0, 0.0)
writeCoord(y=2.0); // (0.0, 2.0)
```

Parallel programs development based on PGAS model: Chapel…

Task parallelism: *begin* operator

- **Syntax**

  begin-stmt:

  begin *stmt*

- **Semantics**
  - Creates a parallel task for **stmt** execution
  - Execution of the parent program is not suspended
- **Example**

```
begin writeln("hello world");
writeln("good bye");
```

Possible outputs

```
(1) hello world      (2) good bye
    good bye             hello world
```

Parallel programs development based on PGAS model: Chapel…

Task parallelism: *sync* type

- **Syntax**

  sync-type:

  sync *type*

- **Semantics**
  - Variables of the **sync** type have two states – «full», «empty»
  - Writing to a **sync** variable changes its state to «full»
  - Reading from a **sync** variable changes its state to «empty»
- **Examples**

```
(1) var lock$: sync bool;
    lock$ = true;
    critical();
    lock$;
```

```
(2) var future$: sync real;
    begin future$ = compute();
    computeSomethingElse();
    useComputeResults(future$);
```

Parallel programs development based on PGAS model: Chapel…

Task parallelism: *cobegin* operator

- **Syntax**

  cobegin-stmt:

  cobegin { *stmt-list* }

- **Semantics**
  - Spawns a parallel task for each operator from the stmt-list
  - Synchronization at the end of block
- **Example**

```
cobegin {
  consumer(1);
  consumer(2);
  producer();
}
```

Parallel programs development based on PGAS model: Chapel…

Task parallelism: *coforall* operator

- **Syntax**

```
coforall-loop:
  coforall index-expr in iterator-expr { stmt }
```

- **Semantics**
  - Spawns parallel tasks for every loop iteration
  - Synchronization at the end of the loop
- **Example** (“Producer-consumer” task)

```
begin producer();
coforall i in 1..numConsumers {
  consumer(i);
}
```

Parallel programs development based on PGAS model: Chapel…

Task parallelism: *atomic* operator

- **Syntax**

  atomic-statement:

  atomic *stmt*

- **Semantics**

  stmt is executed as an atomic operation

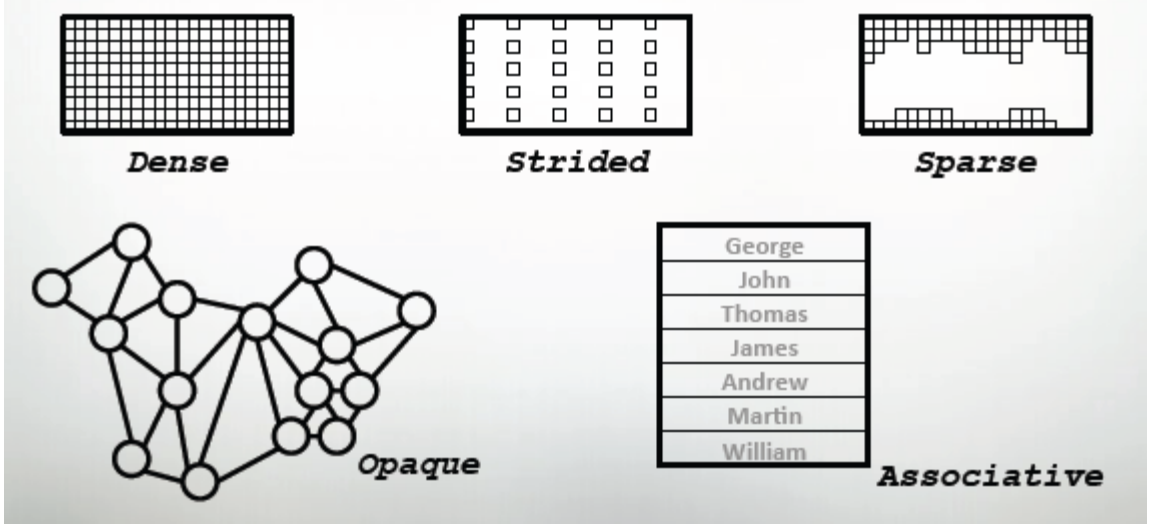
- **Example**

```
atomic A(i) = A(i) + 1;
```

Parallel programs development based on PGAS model: Chapel…

Domains: advanced ways to set ranges

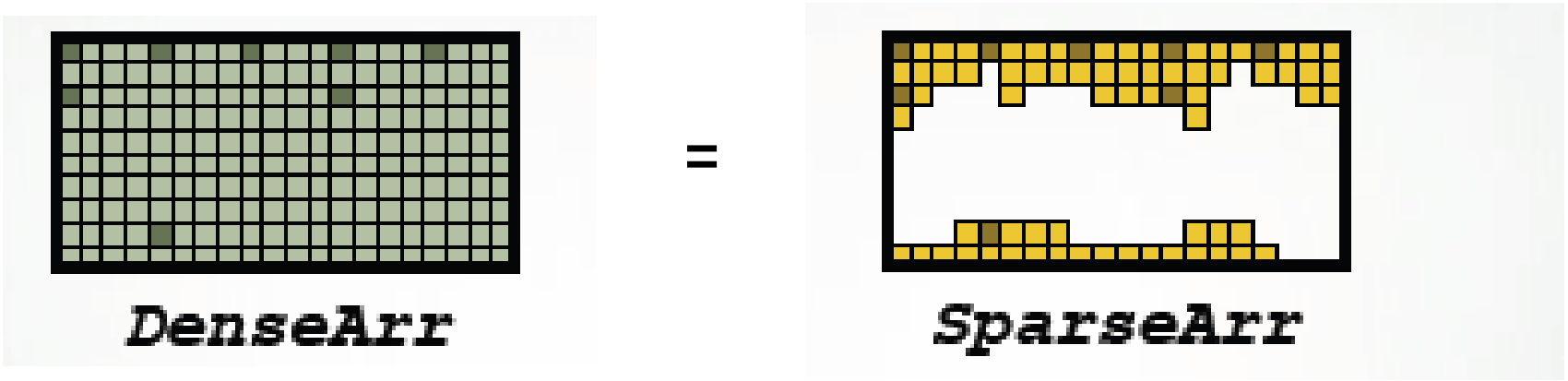
```
var Dense: domain(2) = [1..10, 1..20],
    Strided: domain(2) = Dense by (2, 4),
    Sparse: subdomain(Dense) = genIndices(),
    Associative: domain(string) = readNames(),
    Opaque: domain(opaque);
```



Parallel programs development based on PGAS model: Chapel…

Domain operations

```
forall (i,j) in Sparse {
  DenseArr(i,j) += SparseArr(i,j);
}
```



Parallel programs development based on PGAS model: Chapel…

Data reduction: *reduce* operation

- **Syntax**

  reduce-expr:

  reduce-op ***reduce*** *iterator-expr*

- **Semantics**

  Applies **reduce-op** operation for every data element

- **Example**

```
total = + reduce A;
bigDiff = max reduce [i in InnerD] abs(A(i)-B(i));
```

Parallel programs development based on PGAS model: Chapel…

Data reduction: *scan* operation

- **Syntax**

  scan-expr:

  scan-op ***scan*** *iterator-expr*

- **Semantics**

  Applies **scan-op** operation for every data element (with return of all partial results)

- **Example**

```
var A, B, C: [1..5] int;
A = 1;              // A: 1 1 1 1 1
B = + scan A;       // B: 1 2 3 4 5
B(3) = -B(3);       // B: 1 2 -3 4 5
C = min scan B; // C: 1 1 -3 -3 -3
```

Parallel programs development based on PGAS model: Chapel…

Computing System Model: *locale*

- **Definition**
  - Is an abstraction of computing element
  - Contains a processing unit and memory (storage)
- **Properties**
  - Tasks running within a locale have uniform access to local memory
  - Longer latency for accessing the memory of other locales
- **Example**

  locale – SMP, multicore processor

Parallel programs development based on PGAS model: Chapel…

Computing System Model

- **Declaration of all locales set**

```
config const numLocales: int;
const LocaleSpace: domain(1) = [0..numLocales-1];
const Locales: [LocaleSpace] locale;
```

- **Definition of the locales set at execution start**

```
prompt> a.out --numLocales=8
```

- **Definition of locales set topology**

```
var TaskALocs = Locales[0..1];
var TaskBLocs = Locales[2..numLocales-1];
var Grid2D = Locales.reshape([1..2,1..4]);
```

Parallel programs development based on PGAS model: Chapel…

Computing System Model: Operations

- **Get locale index**

```
def locale.id: int { ... }
```

- **Get locale name**

```
def locale.name: string { ... }
```

- **Get the number of cores on a locale**

```
def locale.numCores: int { ... }
```

- **Get the size of available memory on a locale**

```
def locale.physicalMemory(...) { ... }
```

- **Example – calculating the size of all available memory**

```
const totalSystemMemory =
  + reduce Locales.physicalMemory();
```

Parallel programs development based on PGAS model: Chapel…

Computing System Model: Linking tasks and locales

- **Running tasks on a locale – *on* operation**

```
on-stmt:
  on expr { stmt }
```

- **Semantics**

`stmt` executed on the locale, defined by `expr`

- **Example**

```
var A: [LocaleSpace] int;
coforall loc in Locales do on loc do
  A(loc.id) = compute(loc.id);
```

Parallel programs development based on PGAS model: Chapel…

Computing System Model: Example

```
var x, y: real; // x and y on the locale 0
on Locales(1) { // migrate task to locale 1
  var z: real;  // z on locale 1
  z = x + y;    // remote access to x and y
  on Locales(0) do // return to the locale 0
    z = x + y;  // remote access to z
  // return to the locale 1
  on x do         // transition to the locale 0
    z = x + y;   // remote access to z
  // transition to the locale 1
} // return to the locale 0
```

Parallel programs development based on PGAS model: Chapel…

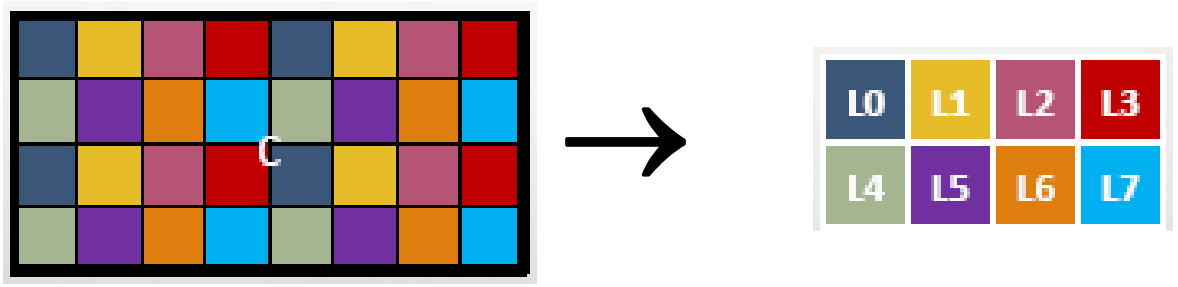
Computing System Model: Data distribution

- (Data) Ranges distribution among over a set of locales

  Example

```
const Dist = new dmap(new Cyclic(startIdx=(1,1)));
varDom: domain(2) dmapped Dist = [1..4, 1..8];
```

Data is distributed on a grid of locales

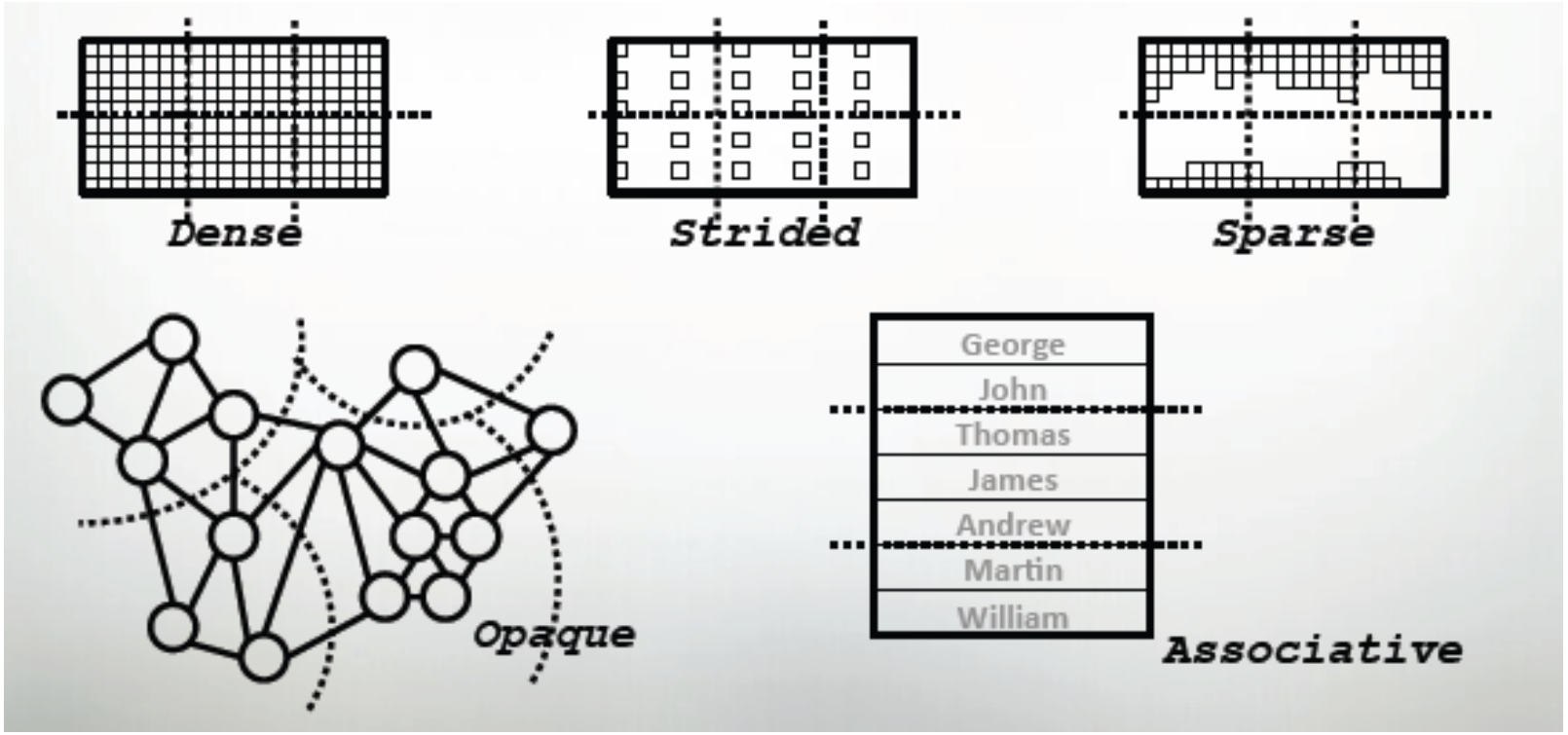


- There is a library of standard distributions
- Possible to define new (own) distributions

Parallel programs development based on PGAS model: Chapel…

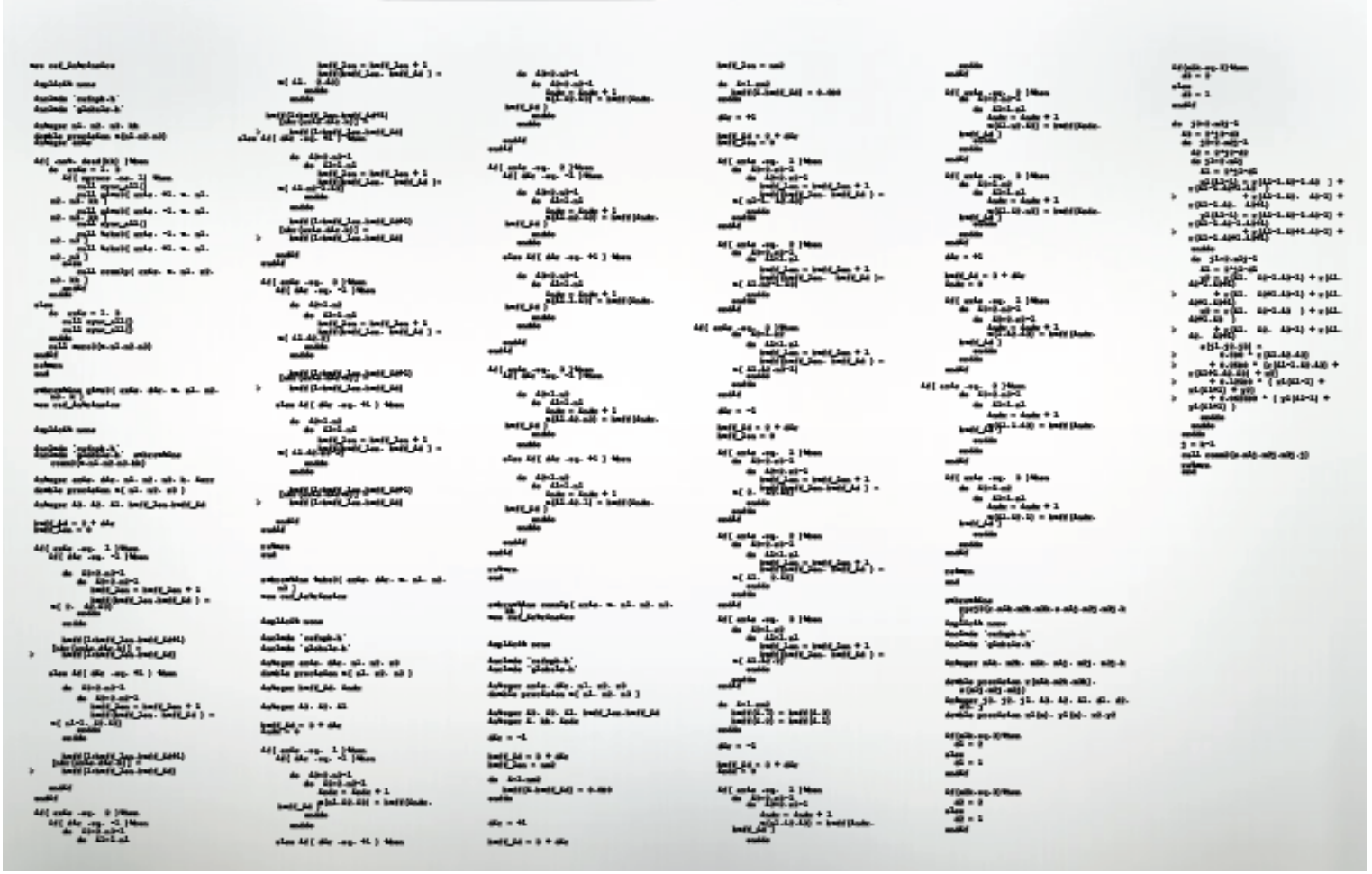
Computing System Model: Data distribution

- Ranges distribution is performed in the same way for every range type



Parallel programs development based on PGAS model: Chapel…

NAG MG Stencil in Fortran+MPI



Parallel programs development based on PGAS model: Chapel…

NAG MG Stencil in Chapel

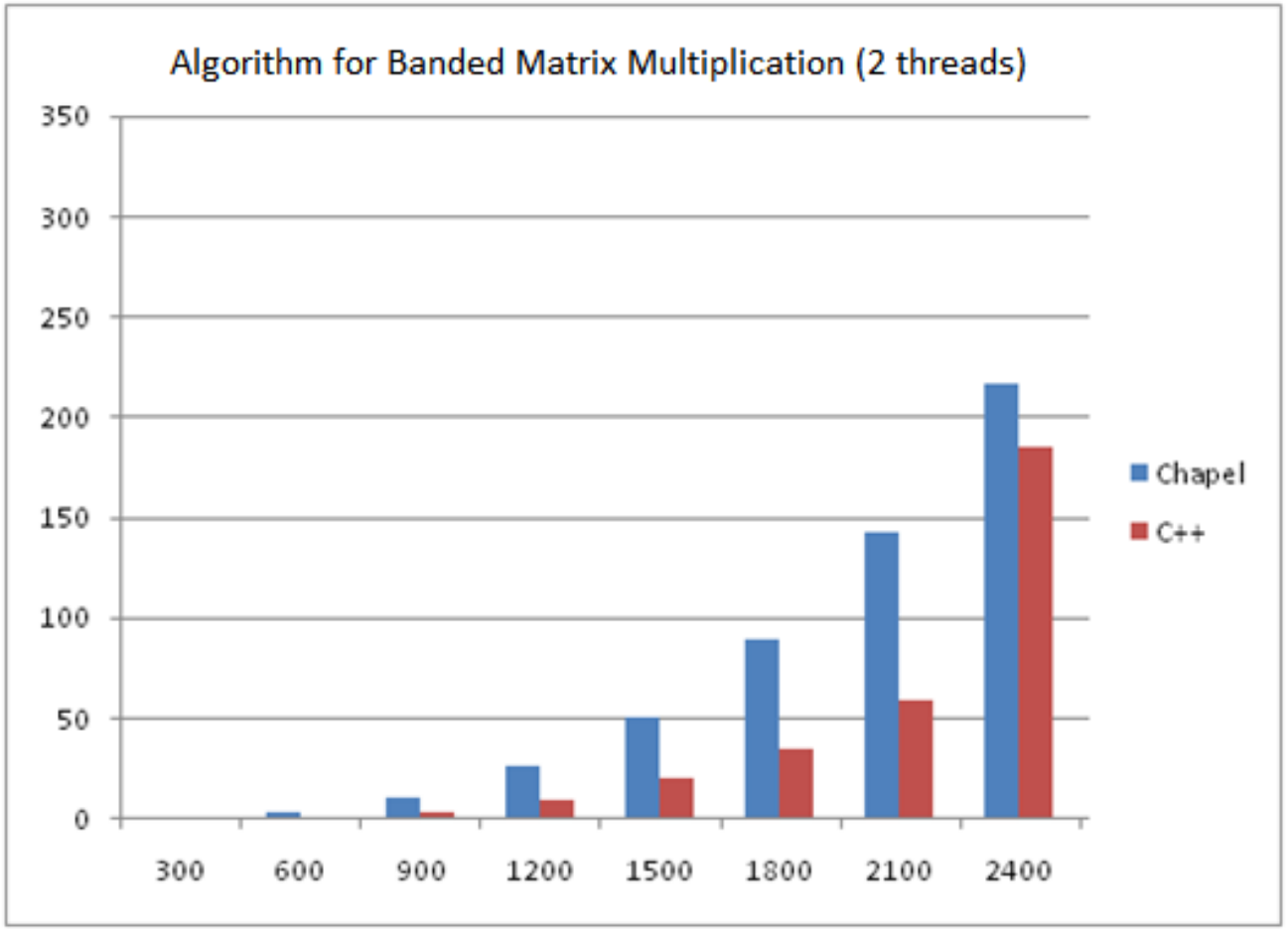
```
def rprj3(S, R) {
  const Stencil = [-1..1, -1..1, -1..1],
    W: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
    W3D = [(i,j,k) in Stencil] W((i!=0)+(j!=0)+(k!=0));

  forall inds in S.domain do
    S(inds) =
      + reduce [offset in Stencil] (W3D(offset) *
          R(inds + offset*R.stride));
```

Parallel programs development based on PGAS model: Chapel

Algorithm for Banded Matrix Multiplication

Intel ®Core™2 DUO T5250 @ 1.5GHz 1.5GHz 2.00 Gb RAM



Conclusion

- **Major approaches to develop parallel programs:**
  - **Usage of libraries** for existing programming languages,
  - **Usage of preprocessor** (directives, comments),
  - **Extension of existing programming languages,**
  - **Creation of new parallel programming languages**
- **Major technologies:**
  - **OpenMP** for systems with shared memory,
  - **MPI** for systems with distributed memory
- **A direction based on the model of a distributed globally-addressed space (PGAS) is actively developed**

Contacts:

Nizhny Novgorod State University

Department of Computational Mathematics and Cybernetics

Victor P. Gergel

gergel@unn.ru

Thank you for attention!

Any questions?